1998

Specifications in software prototyping

Luqi

The Journal of Systems and Software, Vol. 42 (1998), pp. 125-140
http://hdl.handle.net/10945/42329

ELSEVIER

The Journal of Systems and Software 42 (1998) 125–140

The Journal of Systems and Software

Specifications in software prototyping [1]

Luqi [a], Carl K. Chang [b,*], Hong Zhu [c]

[a] Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, USA
[b] Department of EECS (M/C 154), University of Illinois at Chicago, Chicago, IL 60607-7053, USA
[c] Institute of Computer Software, Nanjing University, Nanjing 210093, China

Received 20 December 1996; received in revised form 10 November 1997

Abstract

We explore the use of software specifications for software prototyping. This paper describes a process model for software prototyping, and shows how specifications can be used to support such a process via a cellular mobile phone switch example.

1. Introduction

Classical approaches to software development have been criticized for lack of early feedback about the appropriateness of a proposed system [9]. If testing occurs only after the bulk of the implementation effort, then there is a substantial risk of discovering serious problems near the end of the development project, when it is too late to make major changes. Software prototyping seeks to provide user feedback early in the development process, so that requirements and specifications can be validated before available resources for the initial development have been consumed by implementing an undesirable version of the system.

Early attempts at prototyping have been criticized for lack of documentation and sound engineering. The goal of rapid prototyping is to improve the quality of the systems developed, as measured by their value to their user communities. This implies that prototyping must be coupled with methods for specification and analysis, so that the essential properties of proposed systems can be determined and alternative proposals can be evaluated and compared. Once the desired system behavior is identified, it must be captured in a form that is suitable as a basis for creating and testing the deliverable version of the system. It is unrealistic to assume that the properties of a prototype can be readily discovered in practice just because the prototype can be executed. An execution capabilitymust be augmented with systamatic techniques, theoretical results, and practical automated tools for representing and analyzing the properties of a prototype to fully realize the benefits of rapid prototyping. The types of analysis needed include static symbolic execution techniques as well as generation and execution of particular test cases.

Accurately specifying the desired behavior of a software system before the implementation is developed can be quite difficult in practice. Constructing correct specifications is hard because a set of informal ideas must be turned into a formal model via incomplete and imprecise communication between people. A validation process is needed because correctness of the software depends on whether the specification corresponds to the customers' real needs, in addition to whether the implementation conforms to the specification. Executable prototypes of the specification are useful for obtaining user confirmation that a proposed specification correctly represents the problem, and for guiding the reformulation of the specification in cases where it misrepresents the problem.

This paper explores how software specifications can support formulation of requirements via evolutionary software prototyping. The prototyping process repeats a guess/check/modify cycle until the users agree that the demonstrated behavior is acceptable. Most of the modifications seek to change the behavior of the system to reflect new requirements, rather than to preserve the behavior while improving efficiency.

* Corresponding author. Tel.: +1 312 996 4860; fax: +1 312 413; e-mail: ckchang@eecs.uic.edu.

[1] This research was supported in part by the National Science Foundation under grant number CCR-058453 and by the Army Research Office under grant number ARO 111-95.

0164-1212/98/$19.00
PII: S0164-1212(98)10004-3

The rest of the paper is organized as follows. Section 2 suggests a classification of changes to a software specification to support the prototyping process. Section 3 discusses rapid prototyping and describes the role of specifications in the prototyping process. Section 4 presents a complete example for a cellular mobile phone switch. Section 5 contains conclusions.

2. Software specifications in prototyping

We are still short of software specification methods that are sufficiently powerful to describe all significant properties of a proposed software system and that can represent these properties in a way that is tractable both to practicing software engineers and to decision support tools. For example, algebraic specification methods such as ACT-ONE, Larch, and OBJ3 and denotational methods such as VDM and Z have only limited abilities to express concurrency, synchronization, communication, and real-time constraints. These are central issues in distributed and real-time systems. Specifications would become hard to understand, therefore they deserve formal support that can aid understanding both via concise conceptual models that abstract from low-level details, and via automated analysis and synthesis capabilities. Conversely, CCS and other formalisms that focus on processes in distributed systems are weak in their ability to define abstract data types. Recognition of this issue has led to number of attempts to integrate the approaches, by combining algebraic specification facilities with CSP [19] or Petri nets [18]. Another approach is to extend the actor model of computation and to combine it with a logic for expressing requirements on outputs and state changes [5].

2.1. The event model

The event model is an extension of the actor model that can represent real-time systems as well as concurrent and distributed systems [6]. The primitives of the simplified event model [2] used in this paper are modules, messages, and events.

A module is a black box that interacts with other modules only by sending and receiving messages via interfaces. Modules can represent software systems, such as PSDL operators and types [6], Ada subprograms, tasks and packages; or people and hardware devices.

A message is a data packet that is sent from one module to another. Messages are classified into message types based on the name of the message and signature of the associated data. For example, each overloaded variant of an Ada task entry corresponds to a message type, as does each variant of function or procedure declared in an Ada package specification. Each call of such an entry or subprogram corresponds to an individual message. Messages can also be realized by other mechanisms, such as remote procedure calls, I/O, updates to shared data, exceptions, and hardware interrupts.

An event occurs when a module receives a message at a particular instant of time. Every event has a target module, an arriving message, and an occurrence time, and each event is uniquely identified by these three attributes. Events represent both stimuli and responses, and serve as reference points for timing constraints.

A computation history (or *trace*) is a set of events that is partially ordered by a *causes* relation. The causes relation connects each stimulus event to each of the response events caused by the stimulus.

Requirements expressed in terms of the event model are constraints on the causes relation for all possible traces. Most of these constraints [3] have the form "every event satisfying *pre* must cause a corresponding event satisfying *post*", where the precondition *pre* and the postcondition *post* are predicates on attributes of events and states. The state of module is externally modeled as the sequence of all events that occurred at the module prior to arrival of the most recent stimulus. The events occurring at each module are totally ordered by their occurrence times.

System structures are represented in the event model via the *contains* relation. The contains relation connects each module to each of its subcomponent modules. The contains relation for a hierarchically described system specifies its internal structural relationships. In particular, the contains relation distinguishes the modules inside a system from those outside the system. The image under the contains relation is empty for any module that is primitive at the chosen level of abstraction (see granularity below). A system that is primitive at one level can be viewed at lower levels of abstraction by introducing its subsystems and specifying the interactions between the system and its subsystems. A trace at the higher level of abstraction can be recovered from a lower level trace by removing all events occurring at the subsystems as well as the events at the decomposed system that consist of messages arriving from its subsystems.

2.2. Classification of specification changes

We characterize changes to a system in terms of three orthogonal attributes of the system specification: its vocabulary, its granularity, and its behavior. These attri-

[2] The full event model has an additional primitive for modeling temporal events.

[3] Synchronization constraints have a different form.

butes can be formalized using the event model as follows.

- The *vocabulary* of a system is the set of all external stimuli recognized by the system. An external stimulus is a message that is received by the system and originates from a module outside the system. A stimulus that is recognized by the system is the cause of the associated system response events. The vocabulary is usually infinite, and can be finitely represented as the union of a set of message types. The set of message types recognized by a system can be made finite via a suitable representation for generic message types (e.g. generic subprograms declared in Ada packages). The vocabulary of a system represents the set of functional capabilities provided by the system.
- The *granularity* of a system is the set of all internal stimuli recognized by the system. An internal stimulus is a message that originates from a module inside the system and is received by a module inside the system. The granularity represents the amount of detail in which the computation has been specified. At one extreme is a black-box specification, which does not mention any internal stimuli at all. At the other extreme (if we avoid the embedded hardware) is machine code, which specifies internal events corresponding to individual machine instructions.
- The *behavior* of a system is the set of all possible traces for the system in relation to a given vocabulary and granularity. The behavior of a system contains each of the possible responses of the system to each stimulus in the given vocabulary and granularity. The behavior is usually infinite.

Note that each of these three attributes is a set, and that subset relationships on these attributes are related to refinements. We believe that a useful representation of a design history can be obtained by decomposing each step in the evolution of a software prototype into primitive substeps that preserve two of the three attributes and makes a monotonic change (either $\subset$ or $\supset$) in the third [8].

The part of the behavior that must be preserved by primitive substeps that change the vocabulary or abstraction level is determined by the intersection of the vocabularies of the initial and modified systems, and the intersection of the granularities. This says that changes that add new message types or remove previously defined messages types should not affect the behavior of the system with respect to the other previously defined message types. The proposed restrictions lead to the classification of primitive changes shown in Fig. 1. The symbol A_S represents the attribute A of the original system S, and $A_{S'}$ represents the attribute A of the modified system S'.

Extending, abstracting, refining and constraining changes are meaning preserving, given our convention

Attribute A	Effect of Change	
	$A_S \subset A_{S'}$	$A_S \supset A_{S'}$
Vocabulary	extending	contracting
Granularity	refining	abstracting
Behavior	relaxing	constraining

Fig. 1. Types of changes.

that each primitive change preserves two of the three attributes. Meaning-preserving changes are in general conservative extensions: the new specification satisfies the original one, and it may have additional properties. Contracting and relaxing changes are in general meaning removing, and can be used to construct meaning-changing modifications when combined with the other types of changes. There are two ways to add information to a design: adding message types via extending or refining changes which *add* elements to the vocabulary or granularity, and by adding new constraints via constraining changes which *reduce* the set of legal behaviors.

2.3. Examples of modification types

We illustrate these concepts via examples.

- *Extending changes* add new types of messages to a system. For example, adding a new command to a graphical editor is an extending change because it creates a new message type.
- *Contracting changes* are inverses of extending changes: they remove possible types of interactions from a system. A message type may be dropped because it is deemed useless by customers.
- *Constraining changes* restrict the behavior of a system by placing constraints on legal responses to a stimulus. For example, adding a postcondition to a message specification consisting only of a type signature is a constraining change, as is strengthening a postcondition (replacing the postcondition P with P' where $P' \Rightarrow P \& \neg(P \Rightarrow P')$.
- *Relaxing changes* are inverses of constraining changes. They remove some restrictions on the behavior that exist in the previous version. An important class of relaxing changes is applied because the more restrictive requirement is not logically satisfiable, cannot be implemented with existing technology, or cannot be realized within given constraints on budget and computing resources. An example is relaxing a requirement to keep an airplane exactly on course to the requirement for corrective steering when the airplane strays off its course, thus keeping it within some tolerance of expected position. This type of change is needed because perfect control of physical systems is not feasible.

- *Refining changes* constrain the internal details of the computation specified by a program without changing its vocabulary or its externally visible behavior. Such a change may decompose a module into a network of submodules or choose algorithms and data structures for implementing a module.
- *Abstracting changes* are inverses of refining changes. Abstracting changes abstract away some details of a computation, without affecting the vocabulary or the externally visible behavior. They may occur in reverse engineering process, such as the TMM approach to software maintenance [1].

3. Prototyping via specifications

3.1. The prototyping process

Prototyping enhances communication with the user community by providing an executable model of the system early in the development process. Early feedback from the user community leads to software systems that are more likely to meet user needs, and reduces lifecycle costs because changes made at the early stages of development are much cheaper than changes made after the system has been delivered. Prototyping can also be useful for streamlining software evolution. In this section we discuss specification-based prototyping.

There are two phases in our model of the prototyping process, *prototype evolution* and *production code generation* [15]. The purpose of prototype evolution is to stabilize the software requirements before effort is invested in detailed implementation and optimization. The purpose of production code generation is to generate an efficient implementation when the requirements are stable. If there is a need to modify the requirements after delivery. We can return to prototype evolution, followed by another instance of production code generation. The prototyping process is illustrated in Fig. 2.

The prototype evolution phase consists of the activities labeled "analyze requirements", "construct prototype", and "execute prototype". The process starts with requirements analysis to determine an initial version of the requirements. Next, a prototype is constructed based on the requirements. For complex systems, this process usually requires decomposition of the system into simpler subsystems.

In our approach, a prototype consists of a hierarchy of modules, where each module has behavioral specification augmented with optional implementation information, such as a decomposition or a reusable program. The prototype execution activity demonstrates some typical cases of prototype behavior and generates a series of prototype adjustments based on the customer's quick feedback. This feedback is used in the requirements analysis activity to produce an updated set of requirements and trigger the next prototyping cycle. The analysis traces cases of unwanted behavior to identify incorrect or incomplete requirements, and proposes one or more plausible ways to modify the requirements. The next round of prototype execution tests the validity of the proposed changes, provides guidance to choose between alternative formulations, and explores previously unvalidated aspects of system behavior. This process continues until the requirements have been thoroughly exercised and the customers are satisfied with the demonstrated behavior of the prototype. The result of the prototype evolution phase is a software architecture for the proposed system, which consists of formal specifications for the proposed system, and a system decomposition that includes formal specifications for the subsystems and a description of their interconnections.

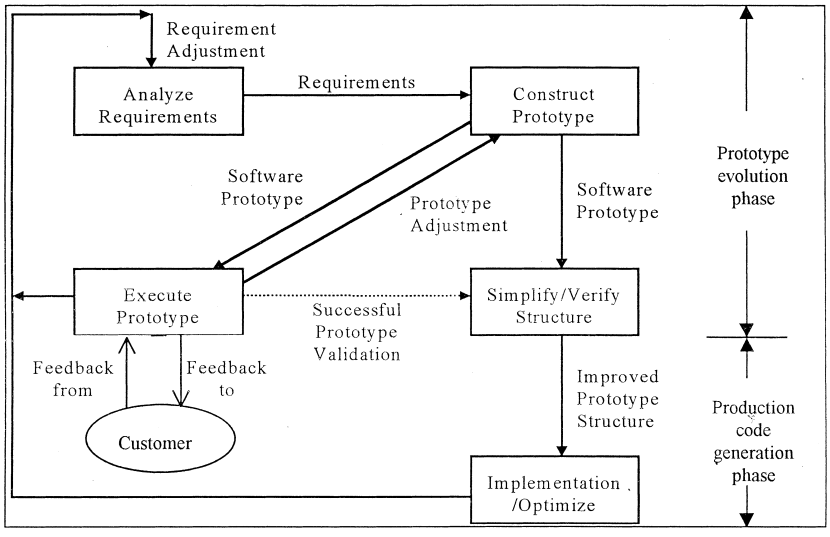

Fig. 2. Prototyping lifecycle process model.

The production code generation phase consists of the activities labeled "simplify/verify structure" and "implement/optimize". The deliverable version of the envisioned system is constructed based on the specifications, decomposition, and other attributes determined in the prototype evolution phase. The simplification of the prototype structure is an optional activity aimed at reducing implementation and maintenance costs. The verification of the structure of the prototype is an optional process whose purpose is to prevent integration problems, especially in cases where different groups or subcontractors produce different subsystems. This verification seeks to prove that the proposed decomposition of a system will meet the specifications for the entire system whenever the subsystems identified in the decomposition meet their specifications. In the near term it is rational to assume that the analysis and design process is potentially imperfect. It may therefore be useful to verify the highest levels of the decomposition before the decomposition is used as a starting point for detailed implementation.

The implement/optimize activity produces efficient implementations of the subsystems. The design-level pragmas associated with the Spec language [6] can in principle provide computer aids for this process. The software to realize this is currently in an early stage of development.

Our procedure for realizing a prototype is illustrated in Fig. 3, which provides an exploded view of the "construct prototype" box in Fig. 2.

For each subsystem in the prototype, the results of requirements analysis are used to propose a system interface and the behavior of the interface is expressed in the Spec language. The specification is then converted to executable form. There are three ways to do this.

- Convert the specification into the executable subset of the Spec language if it is not already in an executable form. This step is necessary because the full Spec language includes unbounded quantifiers and is strong enough to specify functions that are not computable. However, if the requirements are feasible, then the conversion into executable form must also be feasible. In cases where the conversion is too difficult, we can use one of the methods listed below to realize the specification.
- Produce code in a programming language such as Ada. This can be done by retrieving and adapting reusable components based on the specifications [12], or by creating new code, either manually or via program generation and transformation tools.
- Decompose the module into lower-level components. This requires specifying the components (using Spec) and their interconnections (via an augmented data flow diagram, as in PSDL [17]. The prototype design-

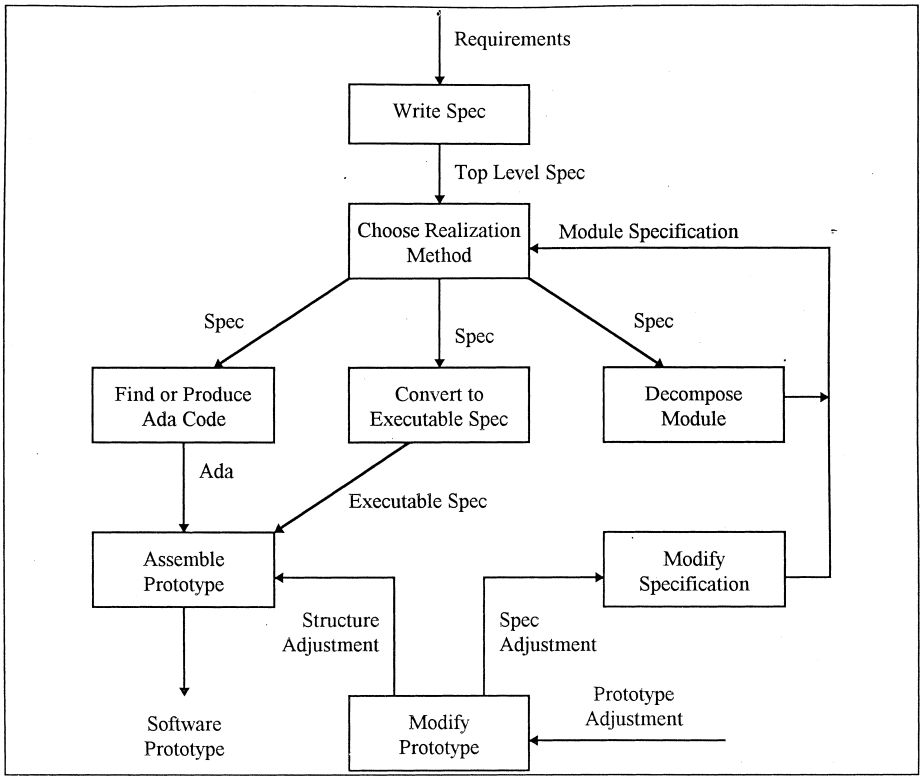

Fig. 3. Constructing a prototype.

er supplies intelligent insights in this step and proposes useful lower-level abstractions. This process can simplify implementation via the previous mechanisms, and can significantly improve performance, especially if frequently executed substeps are realized by efficient reusable program modules.

The result of realizing a subsystem by this mechanism is a hierarchical decomposition into modules that are either directly executable (Ada, C++, etc.) or can be simulated via symbolic execution (Spec, OBJ, etc.).

A key issue for realizing the benefits of prototyping in practice is rapid and correct construction and modification of prototypes via computer aided tools such as the Computer Aided Prototyping System [16].

A prototype demonstration often results in user requests for adjustments to the behavior of the prototype. These adjustments can be classified as Spec adjustments, which modify the specified behavior of a module in the prototype, and as structural adjustments, which rearrange the modulus in the prototype and add or remove subcomponents. Prototype adjustments usually consist of Spec adjustments at the highest levels of the hierarchical structure, and a mixture of Spec adjustments and structural adjustments in the lower levels. One of the goals of computer aid for prototype evolution is to help propagate intended changes from the highest level of the structure to the lower levels, and to ensure that this propagation is complete. Some modifications achieve one goal at the expense of destroying the integrity of the design, so that other modifications must follow to fix things up. Modifications should be packaged together in transactions that guarantee the design to be free from some classes of faults, such as references to undefined variables.

The structure of the prototype must also be periodically simplified to maintain intellectual control and enable future modifications to be carried out with speed and accuracy. This cleanup function involves restructuring the prototype, and removing old features that are no longer needed to support current requirements. It can be considered part of the "modify prototype" activity shown in Fig. 3. This process is generally done between demonstration sessions, based on design reviews or computer-aided dependency analysis rather than on customer feedback. Section 3.2 discusses how specification changes contribute to the construction and evolution of prototypes.

3.2. The role of specifications and changes in prototyping

The initial formulation of a prototype may involve contracting and relaxing changes to the original requirements. These changes are used to simplify the problem and speed up the prototyping process by ignoring less important stimuli, or to relax some of the requirements for stimuli that are modeled in the prototype. Prototyping usually focuses on critical subsystems or particular aspects of the system that cause uncertainty. For example, if human factors are dominant, then formatting requirements may be preserved while requirements on system semantics may be relaxed for the prototype.

Partially developing and then relaxing some of the requirements is often preferable to delaying the elaboration of the less critical requirements, because different aspects of a system are rarely completely independent, although they may be weakly coupled. Access to the complete requirements is sometimes necessary to make sensible choices in a prototyping effort, even if some of those requirements are not intended to be realized by the prototype.

The prototype evolution phase is dominated by a series of nonmonotonic changes to the behavior of the prototype. These changes are realized via contracting and extending changes or via relaxing and constraining changes. Meaning-preserving changes are applied at this stage mainly for adjusting the structure of the prototype to make it easier to understand or modify, and to completely or partially implement non-constructive specifications. Improving efficiency is a major goal only if feasibility of hard real-time constraints must be established or if prototype demonstrations take impractically long to run.

We illustrate some additional uses of meaning-changing changes to realize non-constructive specifications. One of the ways to carry out the "produce Ada code" activity shown in Fig. 3 is to retrieve a reusable software component from a software base, a typical matching activity in the reuse-based prototyping approach. Relaxing changes are useful in this context. If a component that satisfies the specification given by the designer cannot be found, the software base management system can make a query broader by applying relaxing changes to the specification. Some candidates are relaxing changes that drop some of the postconditions. For example, if the output of an operator must be a sequence that contains all elements satisfying a given set of constraints, and which must be in monotonically increasing order with respect to a given ordering, the software base management system can seek modules that satisfy only the first constraint, or only the second constraint. The original specification and two relaxed specifications are shown in Fig. 4.

In this case, the retrieval will succeed for the second relaxed specification, yielding an instance of a generic module that sorts a sequence with respect to an ordering defined by a generic procedure parameter. The retrieved module can then be used to suggest a realization of the original design via decomposition. This example is an instance of a filter decomposition, which succeeds because the postcondition of the first relaxed specification is invariant under the function defined by the second relaxed

```
MESSAGE find_items(s: set{item})
    REPLY (results: sequence{item})
        WHERE ALL(i: item :: i IN results <=>
                    i IN s & size(i) <= max_size),
              sorted{smaller@item}(results)
              -- smaller@item is an ordering on items
```

(A) Original Specification

```
MESSAGE find_items(s: set{item})
    REPLY (results: sequence{item})
        WHERE ALL(i: item :: i IN results <=>
                    i IN s & size(i) <= max_size)
```

(B) First Relaxed Specification

```
MESSAGE find_items(s: set{item})
    REPLY (results: sequence{item})
        WHERE sorted{smaller@item}(results)
         -- smaller@item is an ordering on items
```

(C) Second Relaxed Specification

Fig. 4. Relaxing changes for approximate component retrieval.

specification. Strategies for meeting complex goals via operations that interfere with some subgoals, such as those developed in the context of robot task planning, can be useful for enhancing this approach.

Matching relative to extending and contracting changes is also useful for retrieving reusable software components. Accepting stored components with supertypes for specified arguments, or with optional parameters that were not specified in the query are some simple examples of useful extending changes. Retrieving via partial match with the same postcondition is a useful contracting change that results in a partial operation. This application of changes differs from the use of relaxing changes outlined above, because the changes extend or contract the matching criterion, rather than modifying the query, as did the relaxing changes above.

When integrated with matching, changes can be constructed as part of the matching process, rather than being given a priori. The change that enables the match can contribute to the synthesis of the design. For example, the contracting change used as a guard for the partial operation in a modified decomposition. In general, contracting changes can be applied both to the query and to the stored component. The stored component may not satisfy the query as a total operation, but it may have a partial subfunction that satisfies a contraction of the query as a total operation, but it may have a partial subfunction that satisfies a contraction of the query. An example is using a remainder function to partially satisfy a query seeking a modulo function: the two are identical for non-negative arguments, but differ for negative arguments. The result of the query is a contraction of the remainder operation that is limited to non-negative arguments via a synthesized guard predicate.

Reformulating changes are also useful for speeding up the specification matching process. These changes realize several different normal forms that support signature indexing, fast semantic elimination procedures, and implication checking [21].

Constraining and relaxing changes are useful when the designer discovers that it is possible to implement a stronger version of a component than was required by the original design. A simple example comes form the timing constraints associated with a time-critical operator. Suppose that the original design sought an operator with a maximum execution time of 100 ms, and a software base retrieval located an implementation for the operator with a maximum execution time of only 23 ms. The designer is likely to change the design to require a response for the operator in 23 ms, thus performing a constraining change by replacing a loose timing constraint by a tighter one, as illustrated in Fig. 5.

In this particular application, the constraining change can be followed by a relaxing change that reallocates computation time to some components that are yet to be implemented, making it easier to find implementations for those components. For this example and for other cases involving simple numerical constraints, the associated changes can be efficiently implemented using a constraint maintenance system. More difficult examples include cases where the retrieved component satisfies stronger behavioral properties than were requested by the designer. Such components may subsume functions allocated to other parts of a decomposition, if the designer in fact desired these additional properties. If this is not the case, and the requirements were initially incomplete, then such a retrieval may suggest corresponding constraining changes on the specifications.

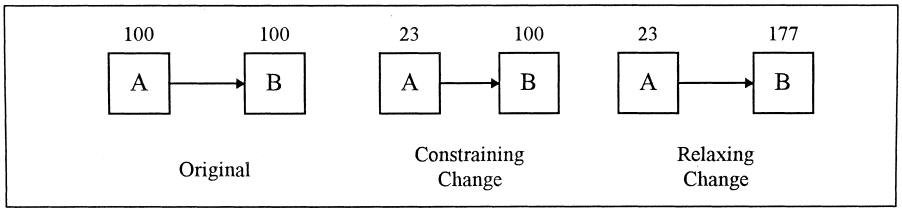

Fig. 5. Timing reallocation induced by software component.

This in turn suggests new requirements to be evaluated by customers in a prototype demonstration session.

In the production code generation phase of the process, the desired behavior of the system is relatively stable, and the major concern is improving efficiency, capacity, or robustness. This part of the process is dominated by meaning-preserving changes for optimizing the design and implementation (the "implement" activity of Fig. 2). Meaning-preserving transformations have been studied extensively [2–4,11,13,14,22]. However, changes to the functionality are sometimes also needed to optimize a design.

Efficient algorithms are often applicable only in special cases, so that their use may constrain the set of problems that can be solved. This makes the operation partially defined. We do not want to leave such an operation partial (the result of contracting change), because that would put the burden on the user to avoid inputs outside the domain of the function. We prefer to remove all constraints on the behavior of the operation outside the domain of the efficient implementation (via a relaxing change) and then to reconstrain the specification by defining safe responses for the remaining cases, such as exception conditions or error messages. A common example of an optimization that speeds up an algorithm by introducing constraints is static memory allocation, which puts a fixed bound on the size of a data structure. Such an optimization adds a class of potential overflow errors to the specification, or at least changes the circumstances under which an overflow error will occur. Since meaning-changing modifications can change the observable behavior of the system, they require a validation step, possibly via an additional prototyping cycle focused on demonstrating the proposed change.

4. Example

In this section, we illustrate the evolution process of software specifications via a case study on simulation software for a cellular mobile phone system [10]. The formal specification is written in the Spec language [6].

In a cellular mobile phone system, many mobile phone units may request communication services via a limited number of channels. A cellular system divides the service region into several service areas (cells) and cluster a certain number of cells as a group in which communication channels are used without repetition. All the other adjacent clusters (areas) can reuse the same spectrum of channels within their own regions without causing signal interference. In this way, maximal efficiency of channel utilization is achieved. There are three types of basic components within a mobile telephone system.

- The Mobile Telephone Units (MTU – a telephone requesting services).
- The Base Site (BS – monitoring MTUs via radio communication).
- The Mobile Switching Center (MSC – the switching administration center for a service area).

The prototype development and evolution process starts with the decomposition of a mobile telephone system into three types of components. At this point we know little about the details of the mobile phones and base sites except that each mobile phone has a unique identity number and a telephone number and each base site also has a unique identity number. This is reflected in the definitions of the `mobile` and `base_site` types. See Fig. 6.

The definition of `mobile_switch_centre` is a syntactically correct skeleton of a specification in Spec. So are the definitions of `mobile_identity`, `base_site_identity` and `dialable_numbers`. Details of these definitions need to be refined by further evolution. Due to lack of space, this paper gives only the details of mobile phone units.

Like an ordinary hard-wired telephone, a mobile phone has a set of digit buttons for dialing telephone or mobile phone numbers. However, a mobile phone also has several additional function keys such as the *on/off switch*, *send button*, *end button*, *erase button*, *answer button*, etc. The *on/off switch* is responsible for powering the mobile phone unit. When the on/off switch is in the on position, the user can initiate a call. The *digit buttons* are designed for dialing phone numbers. Pressing any of these buttons will append the number pressed to the *dialing register*. When the *erase button* is pressed, the dialing register is cleared. When the *send button* is pressed, the number in dialing register is sent to the base site to initiate a call. The *end button* is pressed during a conversation to terminate the phone call and place the

```
TYPE mobile -- Version 0.
    MODEL (id : mobile_identity,
           number: dialable_numbers)
    INVARIANT
        ALL(m1:mobile, m2:mobile ::
            (m1 ~= m2) => (m1.id ~= m2.id & m1.number ~= m2.number))
    MESSAGE create (id: mobile_identity, number: dialable_numbers)
        REPLY(m: mobile)
            WHERE m.id = id, m.number = number
                -- Creates a new mobile unit.
END

TYPE base_site
    MODEL (id: base_site_identity)
    INVARIANT
        ALL(b1:base_site, b2: base_site ::
            (b1 ~= b2) => (b1.id ~= b2.id))
    MESSAGE create (id: base_site_identity)
        REPLY (bs: base_site) WHERE bs.id = id
END

TYPE mobile_switch_center
    -- defines the structure and behavior of the mobile switch center.
END

TYPE mobile_identity
    -- Defines the format of the identity number of mobile phone units.
END

TYPE base_site_identity
    -- Defines the format of the identity number of base sites.
END

TYPE dialable_numbers
    -- Defines the format of the numbers that represent either a
    -- telephone number or a mobile phone number.
END
```

Fig. 6. Top level specification of mobile phone system – Version 0.

phone in the idle on-hook state. When the on/off switch is in the *on* position, the mobile is tuned automatically to listen to the nearest base site's overhead message so that when a call to the mobile phone is paged, it responds to the call by sending a message to base site and *rings* the bell of the mobile to inform the user that a call has been received. The user can then start a conversation by pressing the answer button, which puts the mobile into the active off-hook state. The function of each button is specified by a corresponding message, producing the extended type definition shown in Fig. 7.

Version 1 of `mobile` type definition is an extension of version 0. A number of new message types are added into the vocabulary, but the behavior of a mobile phone is not yet formally specified because some of the state transitions and post-conditions of the messages are described only by informal comments. We have recorded the implicit assumption that mobiles have internal states that are subject to change by inheriting the properties common to all types of mutable object from a standard library module whose definition can be found in [6]. We now perform relaxing/constraining changes that give full specifications of the responses to the messages. The result of these changes is shown in Fig. 8.

When specifying the pre-/post-conditions, it is realized that the mobile unit should keep track of information about the tuned base site. Another message, *tune*, is then added into the type definition of mobile phones. An analysis of the state transitions identifies a trap state, which makes the analyst recognize a missing interaction between the mobile switch center and the mobile unit. A new message *connected* is added to represent this interaction. This results in the extending and constraining changes to the specification shown in Fig. 9.

```
TYPE mobile -- Version 1
    INHERIT Mutable{mobile} -- defines the concept "new".
    MODEL (id : mobile_identity,
           number: dialable_numbers,
           power: enumeration{#on, #off},
           hook : enumeration{#on, #off})
    INVARIANT
        ALL(m1:mobile, m2:mobile ::
            (m1 ~= m2) => (m1.id ~= m2.id & m1.number ~= m2.number))
    MESSAGE create (id: mobile_identity, number: dialable_numbers)
        REPLY(m: mobile)
            WHERE m.id = id, m.number = number,
                  m.power = #off, m.hook = #on
        TRANSITION new(m)
          -- m was not an element of mobile in the previous state
    MESSAGE on_off_switch(m: mobile)
        TRANSITION m.power = toggle(*m.power)
    MESSAGE digit_button(m: mobile, d: digits)
        WHEN m.power = #on
            -- digit buttons are effective only if the power is on.
        TRANSITION -- add the digit d to the end of dialing register.
        OTHERWISE -- do nothing
    MESSAGE erase_button(m: mobile)
        WHEN m.power = #on
        TRANSITION
            -- clear the dialing register.
        OTHERWISE -- do nothing
    MESSAGE send_button(m: mobile)
        WHEN m.power = #on
            -- the 'send' button functions only if the power is on.
        TRANSITION
            -- send out the dialed number and change the state of the
            -- mobile phone unit when the 'send' button is pressed.
        OTHERWISE -- do nothing
    MESSAGE end_button(m: mobile)
        WHEN m.power = #on
        TRANSITION
            -- change the state of the mobile phone unit when the 'end'
            -- button is pressed.
        OTHERWISE -- do nothing
    CONCEPT digits: type
        WHERE subtype(digits, nat), ALL(d: digits :: d <= 9)
    CONCEPT toggle(p: enumeration{#on, #off})
        VALUE(q: enumeration{#on, #off})
        WHERE (p = #on) => (q = #off) & (p= #off) => (q= #on)
END
```

Fig. 7. Specification of mobile phone unit – Version 1.

A type consistency analysis of send_button indicates that sequence{digits} must be a subtype of the previously unspecified type dialable_number. Since we do not have reason to generalize dialable_number (we have not considered the functions of special keys such as * and #), we can provisionally interpret that type to be the same as sequence{digits}. The only other unspecified type is mobile_identity. Since all we know about this type so far is

```
TYPE mobile -- Version 2
  INHERIT Mutable{mobile} -- defines the concept "new".
  MODEL(id : mobile_identity,
        number: dialable_numbers,
        power: enumeration{#on, #off},
        hook: enumeration{#on, #off},
        state: enumeration{#idle, #dialing, #sending, #conversation},
        dialing_register: sequence{digits} )
   INVARIANT
       ALL(m1:mobile, m2:mobile ::
           (m1 ~= m2) => (m1.id ~= m2.id & m1.number ~= m2.number))
   MESSAGE create (id :mobile_identity, number: dialable_numbers)
       REPLY(m: mobile)
           WHERE m.id = id, m.number = number,
                 m.power = #off, m.hook = #on,
                 m.state = #idle, m.dialing_register = []
       TRANSITION new(m)
   MESSAGE on_off_switch(m: mobile)
       TRANSITION m.power = toggle(*m.power)
   MESSAGE digit_button(m: mobile, d: digits)
       WHEN m.power = #on & m.state IN {#idle, #dialing}
       TRANSITION m.dialing_register = *m.dialing_register || [d],
                  m.state = #dialing, m.hook = #off
       OTHERWISE -- do nothing.
   MESSAGE erase_button(m: mobile)
       WHEN m.power = #on & m.state = #dialing
           TRANSITION m.dialing_register = [],
                      m.state = #idle, m.hook = #on
       OTHERWISE -- do nothing.
   MESSAGE send_button(m: mobile)
       WHEN m.power = #on & m.state = #dialing & m.hook = #off
           -- send a message to the base site that the mobile phone unit
           -- is tuned to.
           TRANSITION m.state = #sending
       OTHERWISE -- do nothing.
   MESSAGE end_button(m: mobile)
       WHEN m.power = #on
            & m.state = #conversation & m.hook = #off
           TRANSITION m.state = #idle, m.hook = #on
       OTHERWISE -- do nothing.
   CONCEPT digits: type
           WHERE subtype(digits, nat), ALL(d: digits :: d <= 9)
   CONCEPT toggle(p: enumeration{#on, #off})
       RESULT(q: enumeration{#on, #off})
           WHERE (p = #on => q = #off)&( p = #off => q = #on)
END
```

Fig. 8. Specification of mobile phone unit – Version 2.a.

that it must have an "=" operation, it can be interpreted as any type with a well-defined equality operation. Natural numbers will serve.

Given these interpretations, the above specification becomes executable, because all of the postconditions are free of quantifiers and have the form of equations that unambiguously define the quantities to be constructed in response to each event. Thus in the context of appropriate tool support, we have constructed an executable prototype at this point, which can be tested and demonstrated to clients. The prototype can also be instrumented to check the truth of the invariant after each state transition. In this case the invariant can be made executable by enumeration because the bound

```
MODEL ( ... ...,
        tuned_base_site: base_site,
        signal_strength : nat)
    INVARIANT ...
    ... ...
MESSAGE create (id: mobile_identity, number: dialable_numbers)
    REPLY(m: mobile)
        WHERE m.id = id, m.number = number,
              m.power = #off, m.hook = #on,
              m.state = #idle, m.dialing_register = [],
              tuned_base_site = default_base_site,
              signal_strength = 0
    ... ...
MESSAGE send_button(m: mobile)
    WHEN m.power = #on & m.state = #dialing & m.hook = #off
        SEND calling(n: dialable_number) TO tuned_base_site
          WHERE n = m.dialing_register
        TRANSITION m.state = #sending
    OTHERWISE -- do nothing.
    ... ...
MESSEAGE connected(m: mobile, successful: boolean) -- from MSC
    TRANSITION IF successful
                THEN m.state = #conversation & m.hook = #off
                ELSE m.state = #idle & m.hook = #on
                FI
MESSAGE tune(m: mobile, bs: base_site, strength: nat) -- from BS
    WHEN m.power = #on & m.state ~= #conversation
        TRANSITION
            IF  m.tuned_base_site = bs
            THEN m.signal_strength = strength
            ELSE IF m.signal_strength < strength
            THEN m.tuned_base_site = bs &
                 m.signal_strength = strength
            ELSE true -- do nothing
            FI
    OTHERWISE -- do nothing
CONCEPT default_base_site: base_site -- a constant
```

Fig. 9. Specification of mobile phone unit – Version 3.a.

variables range over a type with a finite population. Any mutable type has a finite population because the instances are dynamically created, and there can be only a finite number of such creation events prior to any given point in a computation.

At this stage it is not useful to demonstrate the prototype to clients because the process of receiving a call is still undefined, and calls cannot yet be completed. However, engineers examining execution traces notice that the mobile is off hook only when it is in the dialing, sending, and conversation states. Monitoring of prototype execution and analysis of the state machine confirm that this is indeed an invariant of the current specification. This suggests checking with the clients whether this is likely to remain true when additional aspects of the application are included in the model. If it is true, then the state component m.hook is not independent: its value can be derived from m.state. Consequently, the specification can be simplified by dropping m.hook from the MODEL and all preconditions and postconditions in this specification. It can be defined as a derived concept if it turns out to be needed for explanation or for specifying future enhancements to the system.

The specification and prototyping of a large and complex software system often divides the task into a number of concurrent processes. Each process concentrates on one particular aspects of the problem. For example, the above specification is only concerned with the

```
MODEL ( ... ...,
        state : enumeration{#idle, #dialing, #sending, #ringing,
                            #conversation},
        ... ... )
    INVARIANT ... ...
MESSAGE paging (m: mobile, page: set(mobile_identity))
    WHEN m.power = #on & m.state = #idle & m.id IN page
        SEND mobile_found(id: mobile_identity) TO tuned_base_site
            WHERE id = m.id

        TRANSITION m.state = #ringing
    OTHERWISE - do nothing
```

Fig. 10. Specification of mobile phone unit – Version 2.b.

initiation process of mobile phone calls. While a team of software engineers works on specifying and prototyping this process of initiating a call and produces the specification given in Fig. 9, another team may well be working on specifying the process of receiving a call.

Based on version 1 of the specification, another set of messages is added to specify the receiving process. The state space extension is also different from the calling process team. The fragments of formal specifications for receiving calls are given in Fig. 10.

When the bell is ringing, the user of the phone can start the conversation by pushing the answer button. Therefore, we add and specify the answer_button message as shown in Fig. 11. This is an extending change.

After the two teams finish their parallel developments, the result versions, i.e. version 3.a and version 3.b, must be merged into a consistent one. It is easy to see that the differences between the two versions are not in conflict with each other. The merging of the two versions results in the specification given in Fig. 12. The whole evolution process is shown in Fig. 13.

As formally specified in Fig. 12, the tuning process is performed when power is on but the state is not conversation. During a conversation, a mobile phone may need re-tuning into another base site if the phone is moving from the coverage area of one base site to another. This process is called handoff. At this stage of specification and prototyping, a software engineer may want to concentrate on the process of tuning, initiating and receiving a call, hence, the refinement of the handoff process is postponed for future development.

Since the mobile phone unit is simple enough so that there is no need to decompose its structure into components, there are no granularity changes during the specification of the first prototype in this example. However, when prototypes are used for deriving the software architecture, granularity changes may happen. Readers are referred to [7] for a theory and method of merging of changes in software structure. There are no timing constraints visible at the black box granularity level of this example. However, they do appear at lower levels of the protocols, which rely on timeout constraints to determine aspects such as the success of failure of a requested connection.

5. Conclusions

Evolutionary prototyping is a promising approach to the problem of formulating system requirements that accurately reflect the needs of the stakeholders. This approach becomes particularly attractive if we can provide formal models and decision support tools to aid the process. One of the reasons software evolution is difficult is that realistic software designs are complicated by optimizations. These optimizations improve the efficiency of the software by introducing additional constraints that complicate the implementation and introduce dependencies between parts of the system that might otherwise be independent.

```
MESSAGE answer_button(m: mobile)
    WHEN m.power = #on  & m.state = #ringing
        SEND start_conversation TO tuned_base_site
        TRANSITION m.hook = #off, m.state = #conversation
    OTHERWISE -- do nothing
```

Fig. 11. Specification of mobile phone unit – Version 3.b.

```
TYPE mobile - Version 4.
   INHERIT Mutable{mobile} -- defines the concept "new".
   MODEL (id: mobile_identity,
          number: dialable_numbers,
          power: enumeration{#on, #off},
          hook: enumeration{#on, #off},
          state: enumeration{#idle, #dialing, #sending, #ringing,
                             #conversation},
          dialing_register: sequence(digits),
          tuned_base_site: base_site,
          signal_strength: nat)
   INVARIANT
       ALL(m1:mobile, m2:mobile ::
           (m1 ~= m2) => (m1.id ~= m2.id) & (m1.number ~= m2.number))
   MESSAGE create (id: mobile_identity, number: dialable_numbers)
       REPLY(m: mobile )
           WHERE m.id = id, m.number = number,
                 m.power = #off, m.hook = #off,
                 m.state = #idle, m.dialing_register = [],
                 tuned_base_site = default_base_site,
                 signal_strength = 0
       TRANSITION new(m)
   MESSAGE on_off_switch(m: mobile)
       TRANSITION m.power = toggle(*m.power)
   MESSAGE digit_button(m: mobile, d: digits)
       WHEN m.power = #on & m.state = #dialing & m.hook = #off
       TRANSITION m.dialing_register = *m.dialing_register || [d]
       OTHERWISE -- do nothing.
   MESSAGE erase_button(m: mobile)
       WHEN m.power = #on
       TRANSITION m.dialing_register = []
           -- clear the dialing register.
       OTHERWISE -- do nothing.
   MESSAGE send_button(m: mobile)
       WHEN m.power = #on & m.state = #dialing & m.hook = #off
           SEND calling(n: dialable_number) TO tuned_base_site
              WHERE n = m.dialing_register
           TRANSITION m.state = #sending
```

Our approach to evolutionary prototyping uses high-level subsystem specifications to simplify evolution support. We have used the prototyping language PSDL to describe the software architecture and the specification language Spec to describe the constraints, obligations, and relationships between the slots in the prototype software architecture. Spec annotations provide a first step towards high level descriptions of optimizations to produce the product versions of the proposed software.

Specification can serve several purposes in the context of prototyping.

- They can be used to document the range of behaviors expected for the component slots in the evolving software architecture.
- They can be used to record the rationale for implementation decisions, to guide later evolution and enhancement of the design.
- They can be used as search keys for retrieving reusable software components to realize subsystems of the prototype.
- They can be used to verify (prove the soundness of) a decomposition before it is used as the basis for division of labor in a detailed implementation and optimization effort. This should reduce the incidence of system integration problems late in the development process.

A continuing challenge is to find the proper balance among expressiveness of specifications, simplicity, and tractability of processing by automated tools. Practical impact of formal specifications hinges on finding formal representations that practical software engineers can understand and are willing to use, and providing tool support with enough practical benefit to overcome the extra effort required to formalize requirements, specifications, and designs.

```
        OTHERWISE·-- do nothing.
    MESSAGE end_button(m: mobile)
        WHEN m.power = #on &
             m.state = #conversation & m.hook = #off
        TRANSITION m.state = #idle
        OTHERWISE -- do nothing.
    MESSEAGE connected(m: mobile, successful: boolean) -- from MSC
        TRANSITION IF successful
                   THEN m.state = #conversation & m.hook = #off
                   ELSE m.state = #idle & m.hook = #on
                   FI
    MESSAGE tune(m: mobile, bs: base_site, strength: nat) -- from BS
        WHEN m.power = #on & m.state ~= #conversation
            TRANSITION
                IF   m.tuned_base_site = bs
                THEN m.signal_strength = strength
                ELSE IF m.signal_strength < strength
                THEN m.tuned_base_site = bs &
                     m.signal_strength = strength
                ELSE true -- do nothing
                FI
        OTHERWISE -- do nothing
    MESSAGE paging (m: mobile, page: set(mobile_identity))
        WHEN m.power = #on & m.state = #idle & m.id IN page
            SEND mobile_found(id: mobile_identity) TO tuned_base_site
                WHERE id = m.id
            TRANSITION m.state = #ringing
        OTHERWISE -- do nothing.
    MESSAGE answer_button(m: mobile)
        WHEN m.power = #on & m.state = #ringing
            SEND start_conversation TO tuned_base_site
            TRANSITION m.hook = #off, m.state = #conversation
        OTHERWISE -- do nothing
    CONCEPT digits: type
            WHERE subtype(digits, nat), ALL(d: digits :: d <= 9)
    CONCEPT toggle(p: enumeration{#on, #off})
        VALUE(q: enumeration{#on, #off})
            WHERE (p = #on => q = #off)&( p = #off => q = #on)
    CONCEPT default_base_site: base_site -- a constant
END
```

Fig. 12. The merging of Version 3.a and Version 3.b – Version 4.

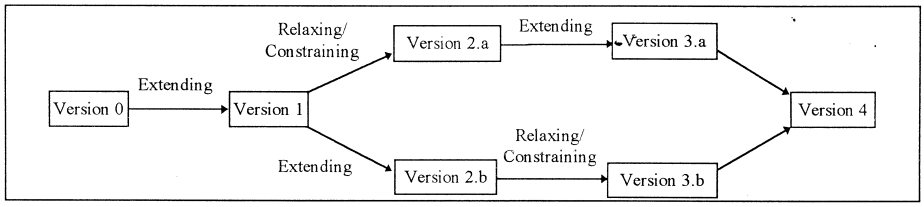

Fig. 13. The evolution process of the specification of mobile phone units.

References

[1] G. Arango, I. Baxter, P. Freeman, C. Pidgeon, TMM: Software Maintenance by Transformation, IEEE Software 3 (3) (1986) 27–39.

[2] F. Bauer et al., The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L, in: G. Goos and J Hartmanis, eds., Lecture Notes in Computer Science, vol. 183, Springer, Berlin, 1985.

[3] F. Bauer et al., The Munich Project CIP. Vol. II: The Program Transformation System CIP-S, in: G. Goos and J Hartmanis, eds., Lecture Notes in Computer Science, vol. 292, Springer, Berlin, 1987.

[4] F. Bauer, B. Moller, H. Partsch, P. Pepper, Formal program construction by transformations – Computer-aided, intuition-

guided programming, IEEE Trans. Software Eng. 15 (2) (1989) 165–180.
[5] V. Berzins, Luqi, An introduction to the specification language spec, IEEE Software 7 (2) (1990) 74–84.
[6] V. Berzins, Luqi, Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada, Addison-Wesley, Reading, MA, 1991.
[7] V. Berzins, D. Dampier, Software merge: Combining changes to decompositions, J. System Integration 6 (1/2) (1996) 135–150.
[8] V. Berzins, Recombining changes to software specifications, in: Proc. SEKE 96, pp. 136–144.
[9] B. Boehm, A spiral model of software development and enhancement, Computer 21 (5) (1988) 61–72.
[10] C.K. Chang, X. Shu, G. Chan, M. Aoyama, An object-oriented real-time distributed simulation of cellular phone switching system, in: Proceedings of 1993 IEEE International Symposium on Circuits and Systems, May 1993, pp. 2232–2235.
[11] S. Fickas, Automating the transformational development of software, IEEE Trans. Software Eng. 11 (11) (1985) 1268–1277.
[12] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, V. Berzins, Software component search, J. Systems Integration (special issue on CAPS) 6 (2) (1996) 93–134.
[13] C. Green, A summary of the PSI Program Synthesis System, in: Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 380–381.
[14] E. Kant, On the efficient synthesis of efficient programs, Artificial Intelligence 20 (3) (1983) 253–305; also appears in: C. Rich, R. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, Los Altos, CA, 1986, pp. 157–183.
[15] Luqi, Software evolution via rapid prototyping, IEEE Comput. 22 (5) (1989) 13–25.
[16] Luqi, M. Ketabchi, A computer aided prototyping system, IEEE Software 5 (2) (1988) 66–72.
[17] Luqi, V. Berzins, R. Yeh, A prototyping language for real-time software, IEEE Trans. Software Eng. 14 (10) (1988) 1409–1423.
[18] B. Kraemer, Luqi, V. Berzins, Compositional semantics of a real-time prototyping language, IEEE Trans. Software Eng. 19 (5) (1993) 453–477.
[19] U. Pletat, Algebraic specification of abstract data types and CCS: An operational junction, in: Protocol Specification, Testing, and Verification, Elsevier, Amsterdam, 1986.
[20] C. Rich, R. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, Los Altos, CA, 1986.
[21] R. Steigerwald, Luqi, J. McDowell, CASE tool for reusable software component storage retrieval in rapid prototyping, Inform. Software Technol. 33 (9) (1991) 698–706.
[22] D. Wile, Program developments: Formal explanations of implementations, Commun. ACM 26 (11) (1983) 902–911; also appears in: C. Rich, R. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, Los Altos, CA, 1986, pp. 191–200.

Chang directs the International Center for Software Engineering (ICSE) at the University of Illinois at Chicago. ICSE is targeted to the research and development of methodologies and tools to help enhance software productivity and quality. Currently Chang is serving as the Secretary of the IEEE Computer Society and an active member of its Executive Committee, after finishing his first 3-year term on the Board of Governors (1995–1997) and being re-elected for the second 3-year term on the BOG (1998–2000). He has been serving on the program committee of International Computer Software and Applications Conference (COMPSAC) since 1982. He served as the Program Chair for the 1993 International Conference on Computer and Information (ICCI) held in Sudbury, Canada. He helped found the International Conference on Requirements Engineering (ICRE) in 1992 and is now chairing the ICRE steering committee. He was the General Chair for ICRE 96. He also served as the Program Co-Chair for COMPSAC 96 held in Seoul, Korea. In 1997 he chaired the Program Committee for the 1997 IEEE Workshop on Future Trends in Distributed Computing Systems (FTDCS) held in Tunis, Tunisia. He has contributed extensively as referees for numerous technical journals including IEEE Transactions on Computers, IEEE Transactions on Software Engineering, IEEE Computer, and IEEE Software, as well as major Conferences such as COMPSAC, ICCL, NCC, International Conference on Parallel Processing (ICPP), and International Conference on Data Engineering. He is on the Editorial Board of IEEE Computer Society Press. He served two terms as the Editor-in-Chief of IEEE Software from 1991–1994. Chang earned a BS in mathematics from National Central University in Taiwan, an MS in computer science from Northern Illinois University, and a Ph. D. in computer science from Northwestern University. Prior to joining UIC, he was with GTE Automatic Electric at Northlake, Illinois (1978–1979) and AT&T Bell Laboratories at Naperville, Illinois (1982–1984). He is a member of ACM, Sigma Xi and Upsilon Pi Epsilon and a senior member of IEEE. His current research interests in software engineering include software process and metrics, object-oriented technology, and distributed real-time systems. He is also working on the critical issues in distributed multimedia computing with applications in Internet-based collaboration technologies. Luqi is a professor of computer science at the US Naval Postgraduate School, Where she chairs the Software Engineering program and leads a team producing highly automated software tools, including CAPS (Computer-Aided Prototyping System). She worked for the Science Academy of China, the computer Center at the University of Minnesota, and industry. Since receiving a Ph. D. in computer science from the University of Minnesota, Luqi has supervised more than eighty graduate student theses. She headed fifty software R&D projects and hundreds of technical papers. She has received support from many organizations including a Presidential Young Investigator Award from the National Science Foundation and the 1997 Technical Achievement Award from the IEEE Computer Society for her research on the enabling technologies for computer-aided prototyping of real-time systems. In addition to chairing or serving on the program committees of more than forty conferences, she is or has been an associate editor for IEEE Expert, IEEE Software, the Journal of Systems Integration, and Design and Process World. She is a senior member of IEEE.